Application-Replay Attack on Java Cards:
When the Garbage Collector Gets Confused

Guillaume Barbu[1,2], Philippe Hoogvorst[1], and Guillaume Duc[1]

[1] Institut Télécom / Télécom ParisTech, CNRS LTCI,
Département COMELEC,
46 rue Barrault, 75634 Paris Cedex 13, France

[2] Oberthur Technologies, Innovation Group,
Parc Scientifique Unitec 1 - Porte 2,
4 allée du Doyen George Brus, 33600 Pessac, France

Abstract. Java Card 3.0 specifications have brought many new features in the Java Card world, amongst which a true garbage collection mechanism. In this paper, we show how one could use this specific feature to predict the references that will be assigned to object instances to be created. We also exploit this reference prediction process in a combined attack. This attack stands as a kind of "application replay" attack, taking advantage of an unspecified behavior of the Java Card Runtime Environment (JCRE) on application instance deletion. It reveals quite powerful, since it potentially permits the attacker to circumvent the application firewall: a fundamental and historical Java Card security mechanism. Finally, we point out that this breach comes from the latest specification update and more precisely from the introduction of the automatic garbage collection mechanism, which leads to a straightforward countermeasure to the exposed attack.

Key words: Java Card, Combined Attack, Garbage Collection, Application Firewall.

1 Introduction

To follow the emergence of new communication technologies, new Java Card specifications have recently been released: Java Card 3.0. This new standard comes in two editions: *Classic* and *Connected*. The *Classic* edition stands as an evolution of previous versions of Java Card. The *Connected* edition represents the real novelty of this version, adding numerous features in the Java Card world such as an embedded web server, the multithreading support, enhanced security policy facilities, an extended API (Application Programming Interface). All along this paper, we will focus on one of the new features introduced by the Java Card 3.0 specifications: a true automatic garbage collection mechanism. This particular feature is one of the rare novelty to be present in both editions of the specifications.

The contribution of this paper is twofold. First, we introduce the concept of reference prediction, taking advantage of the garbage collection. In addition, we show how an attacker with fault injection capacity could, under certain assumptions, use this concept to circumvent the application firewall through a so-called replay attack.

The remainder of this paper is organized as follows: In Section 2, we introduce the principle of reference prediction on Java Card platforms. In Section 3, we describe the application firewall mechanism and expose the application-replay attack under a given implementation assumption. Finally, we discuss the issues raised by the predictability of object references and the different countermeasures that could be implemented in Section 4.

2 Java Card Reference Prediction

This section introduce the notions of Java reference and garbage collection and state the assumptions under which this work is based. Finally, we describe the process we put into practice to achieve this prediction on the tested platforms.

2.1 Reference Assignment

The Java Card Virtual Machine (JCVM) aims at providing an abstraction layer between the hardware device and Java Card applications. This abstraction is the basement of the *write once - run everywhere* philosophy of the Java language. On object instantiation, the JCVM is then responsible for allocating the memory to store this object and assigning it a Java reference, possibly the allocated memory address or a value abstracting this address. Regardless of its exact implementation, we assume in the remainder of this article that these Java references are assigned following a straightforward linear process. That is to say, the next reference to be assigned is the smallest reference that is not already assigned.

Formally, with $(r_i)_{1 \leq i \leq n}$ the previously allocated references, a new reference r_{n+1} is allocated such that:

$$r_{n+1} = \min\{r_i \text{ s.t. } \forall\ r_j < r_i,\ r_j \text{ is used}\} \tag{1}$$

We believe this assumption is not very restrictive since we have successfully tested it on different cards from different manufacturers and different versions of Java Card with the method given in Section 2.3.

For the sake of simplicity we will consider in the remainder of this article that a reference is a value abstracting the physical address of an object. Thus we do not need to consider the size of each object.

2.2 Garbage Collection

The principle of garbage collection is not a novelty, even in the Java Card context. Indeed, Java Card 2.2 proposes (optionally) a memory reclaiming process through the method `JCSystem.requestObjectDeletion()` [1]. However, this method only schedules the object deletion service prior to the next invocation of the `Applet.process()` method. That is to say, unreferenced objects are not actually deleted on the method's call.

The real novelty in the latest version of the Java Card standard is that garbage collection is automatically triggered when memory space becomes insufficient, or on specific event such as card reset for instance. Furthermore, the `System.gc()` [2] method can be called at any time within an application and runs the garbage collector. Unlike the `JCSystem.requestObjectDeletion()` method, when control returns from the `System.gc()` method, the garbage collector should have actually been executed and reclaimed unused memory.

We will not go into further details in the garbage collection mechanism and will only consider that the garbage collector implementation ensures that it will reclaim the memory used by objects that are not accessible anymore (unreferenced). The important point to bare in mind is rather the evolution of the Java Card specifications regarding this functionality.

2.3 Reference Prediction

We can now introduce one of the contribution of our work, the reference prediction process. As this process relies on a particular type confusion, we recall the previous works achieved on this particular topic before giving a complete description of the process.

Previous Works on Type Confusion. Type safety is one of the cornerstone of the Java language security. Most of the literature presenting potential attacks on Java Card (or even on Java SE [3]) use type confusion at some point in the attack path.

Until 2009, attackers have to count on bugs on specific mechanisms or to load an ill-formed application thanks to `.CAP` file manipulation to provoke a type confusion [4–6]. The release of the Java Card 3.0 Connected Edition has rendered this path theoretically impracticable making the On-Card Bytecode Verification (OCBV) of application mandatory. The use of fault attacks has then emerged as a quite efficient technique to reach this point, as exposed in a couple of recent publications [7–12].

In [7], Barbu et al. describe a way to forge object's reference thanks to a type confusion (*e.g.* `Object o = 0x12345678;`). For that matter, they achieve a physical fault injection during a `checkcast` execution in order to render it successful

and provoke a type confusion between two instances of different classes. The first class holding an `Object` class field, and the other an integral class field (`short` or `int` depending on the size of an object's reference). Getting the reference of an object is then as easy as reading an integral field. Similarly, forging the reference of the `Object` field is then as easy as assigning a value to the integral field.

In the remainder of this paper we will consider that an adversary has the ability to read and forge references.

How to get the reference of an object ? The aim of this section is to expose a process to predict the values by which object instances to be created will be referred to. As the previous section lets guess, this process involves the memory allocation and reclaiming mechanisms. But the first requirement for this process is to be able to learn the value of an object's reference.

In the context of Java Card 3.0, this question is answered within the API specification [2] (at least, one answer is suggested).

QUOTE 1. *"As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)"*

This suggestion can be argued by the fact that two instances of the `Object` class will only differ by their internal addresses (or Java references if these two concepts are not merged within the considered platform). It is then consistent to use it to distinguish such objects.

On Java Card 2.x.y platforms (as well as on Java Card 3.0 platforms that do not implement the `hashCode` method as suggested) the following approach including type confusion has to be considered as described in Listing 1.1.

Assuming that it is possible to learn the reference of an object instance appears then reasonable.

How to predict the reference of an object ? Under the linear reference assignment assumption, the prediction process is described by Algorithm 1.

With regards to (1), on step 1 the allocated reference r_k is s.t.

$$r_k = \min\{r_i \text{ s.t. } \forall\ r_j < r_i,\ r_j \text{ is used}\} \tag{2}$$

After, step 2 and 3, we know that r_k is not used anymore.
The following allocated reference r_l will be s.t.

$$r_l = \min\{r_i \text{ s.t. } \forall\ r_j < r_i,\ r_j \text{ is used}\} \tag{3}$$

Listing 1.1. Getting the reference of an object

```
/**
 * Class A holds an Object field: o
 * Class B holds an integral field: ref
 * A a and B b are public fields of the class.
 */
public void initConfusion() {
    A a = new A();
    // Need a fault injection at runtime to avoid
    // a ClassCastException throwing.
    B b = (B) (Object) a;
}
public short getReference(Object o) {
    // Type confusion has "merged" a.o and b.ref
    a.o = o;
    return b.ref;
}
```

Algorithm 1: ReferencePrediction()

0 Delete current unreachable object instances: `System.gc();`

1 Create a new object instance: `Object o = new Object();`

2 Get the reference of this instance: `ref = getReference(o);`

3 Make this object unreachable: `o = null;`

4 Delete this unreferenced object: `System.gc();`

5 The next assigned reference will be `ref`

Consequently, from (2), we know that,

$$r_l = r_k \tag{4}$$

The successive object instantiation and deletion allow us to discover the next available reference, since it is the one that has just been released. Actually, this behavior has been previously observed by Hogenboom et al. [13] in the context of another mechanism leading to memory reclaiming on a Java Card 2.1.1: transaction aborting.

This can be easily tested on any platform supporting the Java Card 3 specifications by running the code in Listing 1.2.

Under the assumptions previously stated, we can now consider that we are able of reading/writing the reference of an object, but also that we can predict the reference of future object instances.

Listing 1.2. Testing the prediction process

```
System.gc();          // First call to the garbage collector to
                      // delete current unreachable references.

o1 = new Object();    // Assign a new reference to o1 and store
h1 = o1.hashCode();   // the value of this reference in h1.

o1 = null;            // Set o1 to null and call the garbage
System.gc();          // collector to actually delete it.

o2 = new Object();    // Assign a new reference to o2 and store
h2 = o2.hashCode();   // the value of this reference in h2.

if (h1 == h2)         // Compare stored references...
    // The assumption is verified.
else
    // The assumption is not verified.
```

3 Application Replay to Circumvent the Application Firewall

This section exposes how the ability of predicting and forging references can be used to circumvent the application context isolation through a so-called replay attack.

3.1 Java Card Context Isolation : the Application Firewall

Java platforms (Java Standard Edition and Java Micro Edition for instance) usually execute one application per virtual machine instance. One difference of the Java Card platform is that it executes a single instance of the virtual machine. Therefore, it must ensure each hosted application that other applications will not access its own data or code within this single virtual machine instance. For that purpose, the specifications mandate the implementation of an application firewall, isolating each application as well as the Java Card Runtime Environment. Figure 1 depicts the context isolation mechanism and the firewall crossing permission in the platform.

Application Firewall Implementation. A possible implementation of the context isolation would be to assign each application group a context identifier. When an application creates an object, this object would then inherit the context identifier of its "maker". Then access across context can be easily checked by comparing the accessing context identifier and the accessed context identifier.

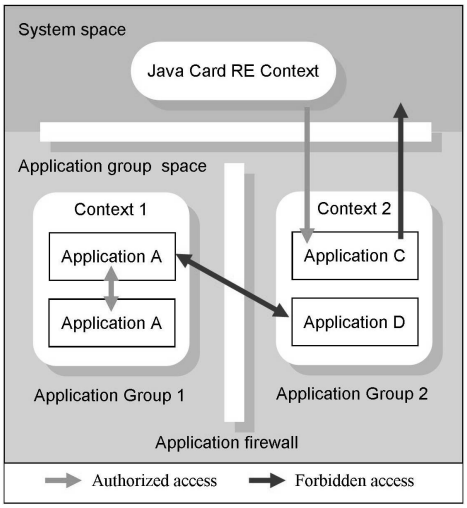


Fig. 1. Contexts within the Java Card platform's object system (as per [14])

If the context identifiers are matching, access is granted, else the firewall deny the access and a `SecurityEcxeption` is thrown.

Such an implementation appears quite suitable in a constrained system. It does not consume too much memory (one 8/16/32-bit word per object to protect it), the access decision is simple (a word comparison) and it does not constrain the number of objects an application can hold. Actually, some experiments based on ill-formed applet loading on Java Cards 2.X.Y from different card manufacturers and on the C reference implementation provided within the Java Card 2.2.2 Development Kit [3] has proven this implementation is (at least has been) used. We will consider such an implementation in the remainder of this article.

This implementation choice leads to a first question:

Question 1. Where does the context identifier comes from ?

Many answers could be given to that question (a random value, an internal counter, the hash of that application's name, ...). However, the important thing is to ensure that two application instances living at the same time in the card does not have the same context identifier. Hence the answer to this question has only a limited interest from an attacker's point of view. We will come back to that point in Section 3.3.

[3] JCDK available at http://download.oracle.com/otn-pub/java/java_card_kit/2.2.2/java_card_kit-2_2_2-linux.zip

3.2 Application Instance Deletion

Java Card platforms allow the post-issuance loading of application. With this capacity comes also that of unloading application. To sum up, an application will then go through the following cycle during its life:

1. Application module loading.
2. Application instance creation.
3. Application execution.
4. Application instance deletion.
5. Application module unloading.

The step we are particularly interested in is the application instance deletion. In order not to depend on any implementation specific mechanism, we will only consider this process according to the specifications.

Java Card 2.2.2. On Java Card 2.2.2, applet instance deletion is processed by an entity referred to as the Applet Deletion Manager (ADM). The behavior of the ADM is specified in the JCRE specification. In particular, it is stated that:

QUOTE 2. *"Applet instance deletion involves the removal of the applet object instance and the objects owned by the applet instance and associated Java Card RE structures."*

Consequently, all objects owned by the applet instance should be actually deleted within the deletion process.

Java Card 3.0. On Java Card 3.0, applet instance deletion is processed by the Card Management Facility (CMF). The behavior of the CMF is specified in the JCRE specification. In particular, it is stated that:

QUOTE 3. *"An application instance is successfully deleted when all objects owned by the application instance are inaccessible on the card."*

QUOTE 4. *"Upon successful deletion, [the card management facility] fires an application instance deletion event - `event:///standard/app/deleted` - with the application instance URI as the event source to all registered listeners."*

The important point to notice here is that what happens between the application instance deletion, the event firing and the actual notification of potential event listeners is not addressed in the specifications. This is indeed the starting point of the attack described in the following section.

3.3 "Application-Replay" Attack on a Java Card 3.0

The previous section has highlighted a difference between the application instance deletion processes on Java Card 2.2.2 and 3.0. Indeed the later does not mandate that object instances belonging to an application instance are deleted

together with the application instance. This lead us to think that the mandatory deletion of objects on Java Card 2.2.X has not been thought as a security mechanism, but rather as a functional one. Actually, since the garbage collection is not mandatory on Java Card 2.2.X platforms, one has to explicitly delete objects that are not used anymore. Else they will still be consuming memory for the whole card lifetime. This explains the disappearing of this statement in the Java Card 3.0 specifications since the garbage collector ensures that those objects will be deleted eventually. The application-replay attack detailed hereafter is then limited to Java Card 3.0 platforms.

The remainder of this section describe a possible attack scenario divided into two steps:

- First, the attacker needs to prevent the deletion of the targeted application's objects;
- Then, the attacker must find a way to access these objects despite the application firewall.

Illegal Memory Consumption. The aim of this first step of the attack is, for the adversary's application, to gain references to objects belonging to the targeted application, even though this application gets deleted.

Let us first put together the different pieces of information gleaned so far. We know from *Quote 3* that the CMF should consider an application instance deletion successful when all objects it owned are inaccessible. This means these objects are garbage-collectable, but not necessarily that they have been garbage-collected. In addition, we know from *Quote 4* that the CMF will fire an event on successful deletion. Finally, we know how to predict and forge object references from Section 2.

Bounding Object Instances Owned by Another Application Instance. Consider now two applications called `Forgery` and `Target`, respectively the adversary's and the targeted application. We assume that `Forgery` is loaded and instantiated. That is to say, its binary representation is on-card and it has been initialized. On the other hand, we assume that that `Target` is only loaded. That is to say its binary representation is on-card but it has not been initialized. Furthermore, we let `Forgery` register an event listener to be notified of application instance deletions (say on the URI `event:///standard/app/deleted/*`).

The `Forgery` application instance can then guess the starting and ending bounds of the `Target` application instance to be, following these steps:

1. Call the garbage collector.
2. Predict the next reference (let us call it `start`).
3. Let `Target` be instantiated.
4. Instantiate an object to get the "current" reference and deduce the last reference instantiated by `Target` (let us call it `end`).

The `Forgery` application then knows that the references of `Target`'s objects are s.t. $\forall r_i \in$ `Target`, `start` $\leq r_i \leq$ `end`.

Preventing the Deletion of Objects on Application Instance Deletion. The attacker can then request the `Target` application instance deletion. During the deletion process, the card management facility will ensure that all objects belonging to this application instance are not referenced anymore. We emphasize the fact that these objects are not necessarily deleted as long as the garbage collector is not executed.

On notification of the application instance successful deletion, the `Forgery` application can then forge object's references in an array of `end` − `start` objects to values between `start` and `end` − 1. This is achieved through a type confusion similar to that exposed in Listing 1.1, considering the confused object is an instance field[4]. By doing so, the attacker prevents these objects from being actually garbage collected, so-to-speak consuming their references.

At this point, the attacker's application instance hold references to objects that do not belong to it. Trying to access these objects in that application would then irremediably lead to a `SecurityException` throwing. The following section adapts the principle of the replay attack to overcome this.

Application Firewall Circumvention. The adversary's application holds references belonging to a deleted application instance. The only way to access these references would then be to collaborate with a new application instance impersonating the deleted one.

It becomes obvious now that the answer to *Question* 1 would then only be useful to help answering the real critical question:

Question 2. Can a new application instance be given the same context identifier as a former (deleted) application instance ?

If we cannot give an accurate answer to *Question* 1 without knowing the exact implementation of the platform, this last question could be answered by experimentation. Given that no Java Card 3.0 platforms have been publicly released so far we have not been able to test this particular behavior on various Java Card 3.0 platforms. Nevertheless, we have run our experimentation on different cards implementing different versions of the Java Card 2 specifications with mostly positive results. Let us assume now that the answer to the last question was "Yes".

The attacker would then only have to instantiate a new application, "send" the forged objects from `Forgery` to that new application and try to access them.

[4] Consequently a single fault injection is necessary for all reference forgeries.

This operation can be repeated until no `SecurityException` is thrown, which means that the new application has been assigned the same context identifier as the original `Target` application instance. That is to say, the new application impersonates the previous `Target`'s application instance.

The last difficulty resides in the "sending" of the forged objects from `Forgery` to the new application, since `Forgery` is not authorized to use these objects by the application firewall. This is why we considered in 3.3 an array of forged objects (the array itself is then still legally usable by `Forgery`). A mere library permits then to store this array from `Forgery` and access its content from the new application without having to pass through the application firewall.

Eventually, the new application instance has then full access to the objects created by the `Target`'s application instance. The application firewall has been circumvented.

So far, this article has proven the possibility to circumvent the Java Card application firewall, under certain assumptions. Nevertheless, the following section shows that this attack can be thwarted with an adequate implementation.

4 Analysis and Countermeasures

The attack describe in the previous section relies on two key elements:

- a "lazy" application instance deletion process.
- the attacker's ability to provoke a type flaw and to forge an object's reference.

Object Deletion. This basement of our attack lies in *Quote* 3, *i.e.* the card manager only ensures that objects owned by the application instance to be deleted are not accessible anymore. In a way, the specifications encourage implementors to give in to the temptation to rely on automatic garbage collection for the effective deletion of these objects.

Thus, it is assumed that the garbage collection will be executed later and that it will delete all inaccessible objects. But between the successful deletion event is fired and the next garbage collection is requested, many things can happen, as exposed within the previous section.
It appears then necessary that the application instance deletion process ensures not only that the objects previously owned by the application to be deleted are inaccessible but also that they are actually deleted when the application instance is deleted. This is indeed what prevents the attack from succeeding on Java Card 2.X platforms, although it seems to us that this has not been specified to enforce security.

This possible breach may be easily taken care of within the implementation of the CMF. Nevertheless, JC3 platforms intending to be considered as secure systems, we go as far as to recommend that mandatory deletion of objects belonging to a deleted application instance should be added in the next update of the Java Card 3.0 specifications.

Ensuring Type Safety. Yet, the type confusion is the technical root of our attack. Without the type confusion, we would have not been able to recover the undeleted objects. Actually, even on platforms supporting OCBV, such a type flaw can be caused by various fault injections. Barbu et al. take advantages of a faulty `checkast` execution, while Vetillard et al. manages to turn an instruction into a `nop`, thus making an instruction from a parameter and provoking an early method return. At CARDIS'11, Barbu et al. presented another way to provoke a type confusion through a faulty operand stack as well as a countermeasure to ensure the operand stack integrity. However this does not prevent the success of the previously cited attacks. Furthermore, other paths to type confusion might be discovered.

The study of the different ways to provoke a type confusion through physical perturbations, as well as the design of countermeasures ensuring type safety in the presence of faults is an ongoing work.

5 Conclusion

In this paper, we have introduced the principle of reference prediction on Java Card platforms and exposed an attack based on the Java Card 3.0 specifications leading to the circumvention of the application firewall.

This work has been based on several assumptions concerning the implementation of the attacked platform which implicitly sketches possible countermeasures (type safety enforcement, actual deletion of objects, unpredictable reference assignment, uniqueness of firewall identifiers). Although we did not put into practice the complete attack path on a Java Card 3.0 platform, we have been able to test successfully the different assumptions required to achieved it on Java Card 2.X platforms from different origins.

Finally, this work outlines a possible weakness in the Java Card 3.0 specifications. Although the exposed attack scenario may appear unlikely on the field, we believe it should be taken into consideration in a future update of the specifications.

References

1. Sun Microsystems Inc.: Application Programming Interface, Java Card Platform Version 2.2.2 (2006)

2. Sun Microsystems Inc.: Application Programming Interface, Java Card Platform Version 3.0.1 Connected Edition (2009)
3. Govindavajhala, S., Appel, A.W.: Using Memory Errors to Attack a Virtual Machine. In: SP'03 : Proceedings of the 2003 IEEE Symposium on Security and Privacy, Washington, DC (2003) 154
4. Witteman, M.: Java Card Security. In: Information Security Bulletin. Volume 8. (2003) 291–298
5. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Smart Card Research and Advanced Application Conference (CARDIS08). LNCS, Springer Verlag (2008) 1–16
6. Iguchi-Cartigny, J., Lanet, J.L.: Developping a Trojan Applet in a Smart Card. Journal in Computer Virology (2010)
7. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card Combining Fault and Logical Attacks. In: Smart Card Research and Advanced Application Conference (CARDIS10). Volume 6035 of LNCS., Springer Verlag (2010) 148–163
8. Vetillard, E., Ferrari, A.: Combined Attacks and Coutermeasures. In: Smart Card Research and Advanced Application Conference (CARDIS10). Volume 6035 of LNCS., Springer Verlag (2010) 133–147
9. Sere, A., Lanet, J.L., Iguchi-Cartigny, J.: Checking the Paths to Identify Mutant Application on Embedded Systems. In: SecTech 2010, International Conference on Security Technology. Volume 6485 of LNCS., Springer Verlag (2010) 459–468
10. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In: Smart Card Research and Advanced Application Conference (CARDIS11), to be published. (2011)
11. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Smart Card Research and Advanced Application Conference (CARDIS11), to be published. (2011)
12. Sere, A., Lanet, J.L., Iguchi-Cartigny, J.: Evaluation of Countermeasures Against Fault Attacks on Smart Cards. International Journal of Security and Its Applications (**5**) 49–61
13. Hogenboom, J., Mostowski, W.: Full memory read attack on a java card. In: 4th Benelux Workshop on Information and System Security Proceedings (WISSEC'09). (2009)
14. Sun Microsystems Inc.: Runtime Environment Specification, Java Card Platform Version 3.0.1 Connected Edition. (2009)